Programming 8-bit PIC Microcontrollers in C

Martin Bates
Elsevier 2008

This presentation contains illustrations from the book 'Programming 8-bit PIC Microcontrollers in C' by Martin Bates

Part 1 Microcontroller Systems
describes in detail the internal architecture and interfaces available in the PIC 16F887A, a typical PIC chip, as well as outlining the main features of the development system

Part 2 C Programming Essentials
provides simple example programs for the microcontroller which show the basic principles of C programming, and interfacing to basic I/O devices

Part 3 C Peripheral Interfaces
provides example programs for operating PIC chips with a full range of peripherals, using timers and interrupts

Part 2

C PROGRAMMING ESSENTIALS

Outline

2.1 PIC16 C Getting Started
- Simple program and test circuit
- Variables, looping, and decisions
- SIREN program

2.2 PIC16 C Program Basics
- Variables
- Looping
- Decisions

2.3 PIC16 C Data Operations
- Variable types
- Floating point numbers
- Characters
- Assignment operators

2.4 PIC16 C Sequence Control
- While loops
- Break, continue, goto
- If, else, switch

2.5 PIC16 C Functions and Structure
- Program structure
- Functions, arguments
- Global and local variables

2.6 PIC16 C Input and Output
- RS232 serial data
- Serial LCD
- Calculator and keypad

2.7 PIC16 C More Data Types
- Arrays and strings
- Pointers and indirect addressing
- Enumeration

2.1 PIC16 C Getting Started

- Simple program and test circuit
- Variables, looping, and decisions
- SIREN program

Programming PIC microcontrollers in C is introduced here using the simplest possible programs, assuming that the reader has no previous experience of the language.

The CCS compiler uses ANSI standard syntax and structures. However, a compiler for any given microcontroller uses its own variations for processor-specific operations, particularly input and output processes. These are fundamental to MCU programs and so will be introduced from the start.

Simple Program

Microcontroller programs contain three main features:

- Sequences of instructions
- Conditional repetition of sequences
- Selection of alternative sequences

The following basic programs show how these processes are implemented in CCS C. The program in Listing 2.1 is a minimal program that simply sets the bits of an 8-bit port in the 16F877 to any required combination.

Listing 2.1 A program to output a binary code

```
/*  Source code file:               OUTNUM.C
    Author, date, version:          MPB  11-7-07  V1.0
    Program function:               Outputs an 8-bit code
    Simulation circuit:             OUTBYTE.DSN
*******************************************************/

#include "16F877A.h"                // MCU select

void main()                         // Main block
{
    output_D(255);                  // Switch on outputs
}
```

Figure 2.1 MPLAB IDE Screenshot

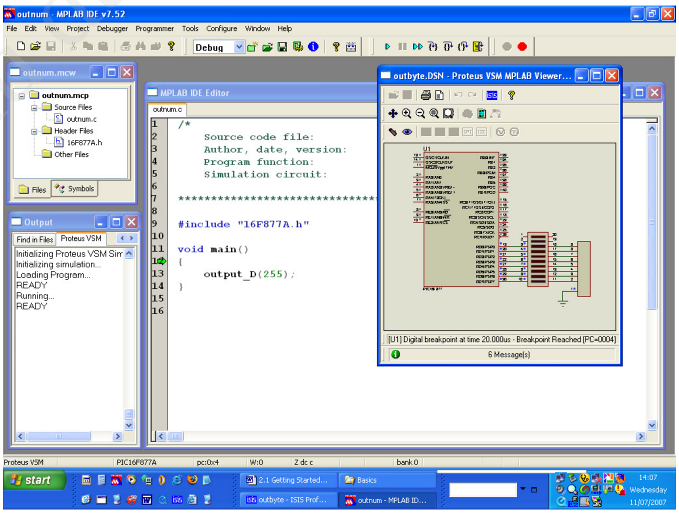


Figure 2.2 ISIS dialogue to attach program

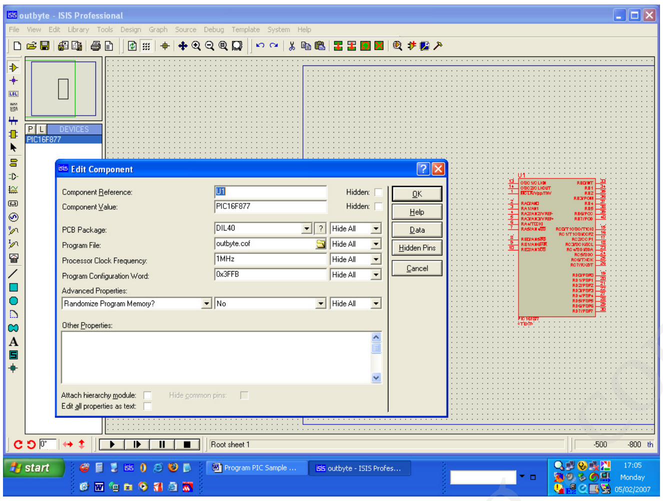
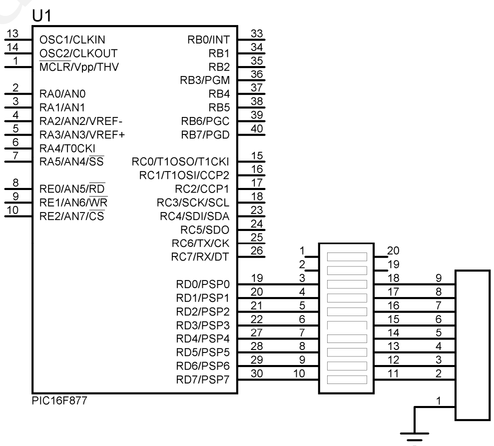


Figure 2.3 OUTBYTE.DSN test circuit with output LEDs



2.2 PIC16 C Program Basics

PIC16 C Program Basics
- Variables
- Looping
- Decisions

The purpose of an embedded program is to read in data or control inputs, process them and operate the outputs as required. Input from parallel, serial, and analog ports are held in the file registers for temporary storage and processing; and the results are output later on, as data or a signal.

The program for processing the data usually contains repetitive loops and conditional branching, which depends on an input or calculated value.

Variables

Most programs need to process data in some way, and named variables are needed to hold their values. A variable name is a label attached to the memory location where the variable value is stored.

In C, the variable label is automatically assigned to the next available location or locations (many variable types need more than 1 byte of memory). The variable name and type must be declared at the start of the program block, so that the compiler can allocate a corresponding set of locations.

Variable values are assumed to be in decimal by default; so if a value is given in hexadecimal in the source code, it must be written with the prefix 0x, so that 0xFF represents 255, for example.

A variable called x is used in the program in Listing 2.2 , VARI.C. Longer labels are sometimes preferable, such as " output_value, " but spaces are not allowed. Only alphanumeric characters (a–z, A–Z, 0–9) and underscore, instead of space, can be used.

By default, the CCS compiler is *not case sensitive, so 'a' is the same as 'A' (even though* the ASCII code is different). A limited number of key words in C, such as main and include , must not be used as variable names.

Listing 2.2 Variables

```
/*  Source code file:          VARI.C
    Author, date, version:     MPB  11-7-07  V1.0
    Program function:          Outputs an 8-bit variable
    Simulation circuit:        OUTBYTE.DSN
*******************************************************/

#include "16F877A.h"

void main()
{
    int x;                  // Declare variable and type

    x=99;                   // Assign variable value
    output_D(x);            // Display the value in binary
}
```

Looping

Most real-time applications need to execute continuously until the processor is turned off or reset. Therefore, the program generally jumps back at the end to repeat the main control loop. In C this can be implemented as a " while " loop, as in Listing 2.3 .

The condition for continuing to repeat the block between the while braces is contained in the parentheses following the while keyword.

The block is executed if the value, or result of the expression, in the parentheses is not zero. In this case, it is 1, which means the condition is always true; and the loop repeats endlessly.

This program represents in simple form the general structure of embedded applications, where an initialization phase is followed by an endless control loop. Within the loop, the value of x is incremented (x ++) . The output therefore appears to count up in binary when executing. When it reaches the maximum for an 8-bit count (11111111 255), it rolls over to 0 and starts again.

Listing 2.3 Endless loop

```
/*  Source code file:              ENDLESS.C
    Author, date, version:         MPB  11-7-07  V1.0
    Program function:              Outputs variable count
    Simulation circuit:            OUTBYTE.DSN
*******************************************************/

#include "16F877A.h"

void main()
{
       int x;                      // Declare variable

       while(1)                    // Loop endlessly
       {      output_D(x);         // Display value
              x++;                 // Increment value
       }
}
```

Decision Making

The simplest way to illustrate basic decision making is to change an output depending on the state of an input. A circuit for this is shown in Figure 2.4 , INBIT.DSN. The switch generates an input at RC0 and RD0 provides the test output.

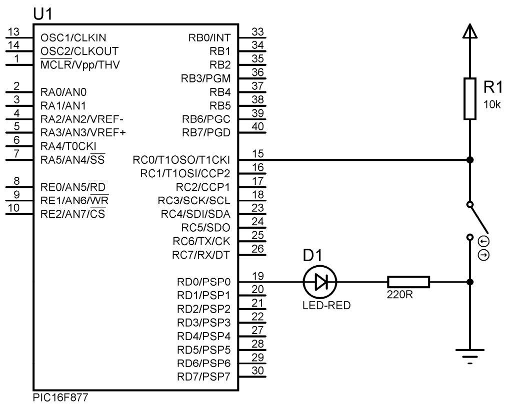
The common keyword for selection in many high level languages is IF. Program IFIN.C (Listing 2.4) has the usual endless " while " loop but contains a statement to switch off Port D initially.

The input state is read within the loop using the bit read function input(PIN_C0). This assigns the input value 1 or 0 to the variable x. The value is then tested in the if statement and the output set accordingly.

Note that the test uses a double equals to differentiate it from the assignment operator used in the previous statement.

The effect of the program is to switch on the output if the input is high. The switch needs to be closed before running to see this effect. The LED cannot be switched off again until the program is restarted.

Figure 2.4 INBIT.DSN test circuit with input switch



Listing 2.4 IF statement

```
/*  Source code file:           IFIN.C
    Author, date, version:      MPB  11-7-07  V1.0
    Program function:           Tests an input
    Simulation circuit:         INBIT.DSN
*******************************************************
/

#include "16F877A.h"

void main()
{
    int x;                  // Declare test var.
    output_D(0);            // Clear all outputs

    while(1)                // Loop always
       {
        x = input(PIN_C0);              // Get input
        if(x==1)output_high(PIN_D0);    // Change out
       }
}
```

Loop Control

The program can be simplified by combining the input function with the condition statement as follows:

if (input(PIN_C0)) output_high(PIN_D0);

The conditional sequence can also be selected by a while condition. In Program WHILOOP.C (Listing 2.5), the input is tested in the loop condition statement and the output flashed on and off while the switch is open (input high). If the switch is closed, the flash loop is not executed and the LED is switched off.

The program also demonstrates the ***delay function***. If this were absent, the loop would execute in just a few microseconds, since each machine code instruction takes 4 μ s at a clock rate of 1 MHz.The flashing of the output would be invisible. The delay required (in milliseconds) is given as the function parameter, and a reference to the function library is provided at the start of the program with the **# use** directive. This allows the compiler to find the library routine **delay_ms()** . The clock speed of the target processor must be given in the use directive, so that the correct delay is calculated within the function.

Listing 2.5 Conditional loop

```
/*  Source code file:         WHILOOP.C
    Author, date, version:    MPB  11-7-07  V1.0
    Program function:         Input controls output loop
    Simulation circuit:       INBIT.DSN
*******************************************************/

#include "16F877A.h"
#use delay (clock=1000000)    // MCU clock = 1MHz

void main()
{
  while(1)
  {
    while(input(PIN_C0));     // Repeat while switch open
    {   output_high(PIN_D0);
        delay_ms(300);        // Delay 0.3s
        output_low(PIN_D0);
        delay_ms(500);        // Delay 0.5s
    }
    output_low(PIN_D0);       // Switch off LED
  }
}
```

FOR Loop

The WHILE loop repeats until some external event or internally modified value satisfies the test condition. In other cases, we need a loop to repeat a fixed number of times. The FOR loop uses a loop control variable, which is set to an initial value and modified for each iteration while a defined condition is true. In the demo program FORLOOP.C (Listing 2.6), the loop control parameters are given within the parentheses that follow the for keyword.

The loop control variable x is initially set to 0, and the loop continues *while it is less than 6.* Value x is incremented each time round the loop. The effect is to flash the output five times.

The FORLOOP program also includes the use of the while loop to wait for the switch to close before the flash sequence begins. In addition, an unconditional while loop terminates the program, preventing the program execution from running into undefined locations after the end of the sequence. This is advisable whenever the program does not run in a continuous loop. Note that the use of the empty braces, which contain no code, is optional.

Listing 2.6 FOR Loop

```
// FORLOOP.C  Repeat loop a set number of times

#include "16F877A.h"
#use delay (clock=1000000)

void main()
{
    int x;

    while(input(PIN_C0)){};       // Wait until switch closed

    for (x=0; x<5; x++)           // For loop conditions
    {
      output_high(PIN_D0);        // Flash sequence
      delay_ms(500);
      output_low(PIN_D0);
      delay_ms(500);
    }
    while(1);                     // Wait for reset
}
```

SIREN Program

A program combining some of these basic features is shown in SIREN.C (Listing 2.7). This program outputs to a sounder rather than an LED, operating at a higher frequency.
The delay is therefore in microseconds. The output is generated when the switch is closed (input C0 low). The delay picks up the incrementing value of " step, " giving a longer pulse each time the for loop is executed. This causes a burst of 255 pulses of increasing length (reducing frequency), repeating while the input is on.

Note that 255 is the maximum value allowed for " step, " as it is an 8-bit variable. When run in VSM, the output can be heard via the simulation host PC sound card. Note the inversion of the input test condition using ! not true.
The header information is now more extensive, as would be the case in a real application. Generally, the more complex a program, the more information is needed in the header. Information about the author and program version and/or date, the compiler version, and the intended target system are all useful. The program description is important, as this summarizes the specification for the program.

Listing 2.7 Siren Program

```
/*  Source code file:            SIREN.C
    Author, date, version:       MPB  11-7-07  V1.0
    Program function:            Outputs a siren sound
    Simulation circuit:          INBIT.DSN
 *******************************************************/

#include "16F877A.h"
#use delay (clock=1000000)

void main()
{
int step;

  while(1)
  {
    while(!input(PIN_C0))                // loop while switch ON
    {
       for(step=0;step<255;step++)       // Loop control
       {
          output_high(PIN_D0);           // Sound sequence
          delay_us(step);
          output_low(PIN_D0);
          delay_us(step);
       }
     }
   }
}
```

Listing 2.8 Program Blank

```
/*      Source Code Filename:
        Author/Date/Version:
        Program Description:
        Hardware/simulation:
****************************************************************/

#include "16F877A.h"            // Specify PIC MCU
#use                            // Include library routines

 void main()                    // Start main block
{
        int                     // Declare global variables

        while(1)                // Start control loop
        {
                                // Program statements
        }
}                               // End main block
```

Blank Program

A blank program is shown in Listing 2.8 , which could be used as a general template.

We should try to be consistent in the header comment information, so a standard comment block is suggested. Compiler directives are preceded by hash marks and placed before the main block. Other initialization statements should precede the start of the main control loop. Inclusion of the unconditional loop option while(1) assumes that the system will run continuously until reset.

We now have enough vocabulary to write simple C programs for the PIC microcontroller.

A basic set of CCS C language components is shown in Table 2.1 . Don't forget the semicolon at the end of each statement.

Table 2.1 A basic set of CCS C components

Compiler Directives

`#include source files`	Include another source code or header file
`#use functions(parameters)`	Include library functions

C Blocks

`main(condition) {statements }`	Main program block
`while(condition) {statements }`	Conditional loop
`if(condition) {statements }`	Conditional sequence
`for(condition) {statements }`	Preset loop

C Functions

`delay_ms(nnn)`	Delay in milliseconds
`delay_us(nnn)`	Delay in microseconds
`output_x(n)`	Output 8-bit code at Port X
`output_high(PIN_nn)`	Set output bit high
`output_low(PIN_nn)`	Set output bit low
`input(PIN_nn)`	Get input

2.3 PIC16 C Data Operations

- Variable types
- Floating point numbers
- Characters
- Assignment operators

A main function of any computer program is to carry out calculations and other forms of data processing. Data structures are made up of different types of numerical and character variables, and a range of arithmetical and logical operations are needed.

Microcontroller programs do not generally need to process large volumes of data, but processing speed is often important.

Table 2.1 Integer Variables

Name	Type	Min	Max
`int1`	1 bit	0	1
`unsigned int8`	8 bits	0	255
`signed int8`	8 bits	-127	+127
`unsigned int16`	16 bits	0	65525
`signed int16`	16 bits	-32767	+32767
`unsigned int32`	32 bits	0	4294967295
`signed int32`	32 bits	-2147483647	+2147483647

Table 2.2 Microchip/CCS Floating Point Number Format

Exponent	Sign	Mantissa
xxxx xxxx	x	xxx xxxx xxxx xxxx xxxx xxxx
8 bits	1	23 bits

Table 2.4 Example of 32-bit floating point number conversion

```
FP number:     1000 0011 1101 0010 0000 0000 0000 0000

Mantissa:                 101 0010 0000 0000 0000 0000

Exponent:      1000 0011

Sign:                    1 = negative number
```

Figure 2.5 Variable Types

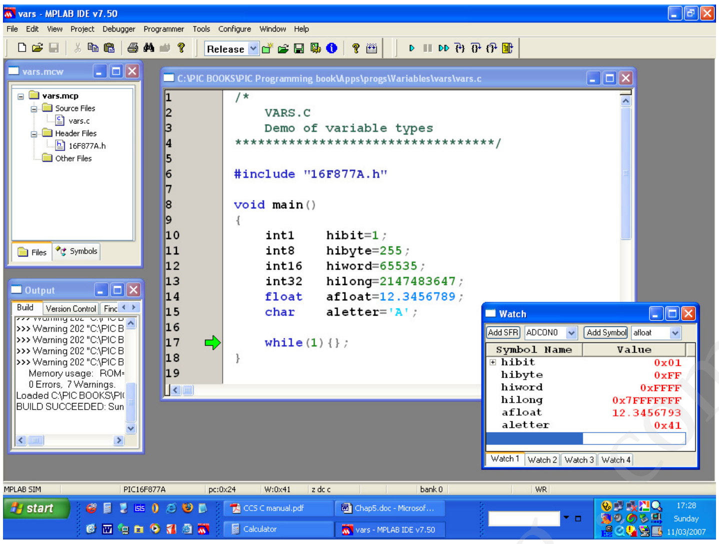


Table 2.5 ASCII Codes

Low Bits	High Bits					
	0010	0011	0100	0101	0110	0111
0000	Space	0	@	P	`	p
0001	!	1	A	Q	a	q
0010	"	2	B	R	b	r
0011	#	3	C	S	c	s
0100	$	4	D	T	d	t
0101	%	5	E	U	e	u
0110	&	6	F	V	f	v
0111	'	7	G	W	g	w
1000	(	8	H	X	h	x
1001	)	9	I	Y	i	y
1010	*	:	J	Z	j	z
1011	+	;	K	[	k	{
1100	,	<	L	\	l	|
1101	-	=	M	]	m	}
1110	.	>	N	^	n	~
1111	/	?	O	_	o	Del

Table 2.6 Arithmetic and Logical Operations

OPERATION	OPERATOR	DESCRIPTION	SOURCE CODE	EXAMPLE	RESULT
Single operand					
Increment	++	*Add one to integer*	`result = num1++;`	0000 0000	0000 0001
Decrement	--	*Subtract one from integer*	`result = num1--;`	1111 1111	1111 1110
Complement	~	*Invert all bits of integer*	`result = ~num1;`	0000 0000	1111 1111
Arithmetic Operation					
Add	+	*Integer or Float*	`result = num1 + num2;`	0000 1010 + 0000 0011	0000 1101
Subtract	-	*Integer or Float*	`result = num1 - num2;`	0000 1010 − 0000 0011	0000 0111
Multiply	*	*Integer or Float*	`result = num1 * num2;`	0000 1010 * 0000 0011	0001 1110
Divide	/	*Integer or Float*	`result = num1 / num2;`	0000 1100 / 0000 0011	0000 0100
Logical Operation					
Logical AND	&	*Integer Bitwise*	`result = num1 & num2;`	1001 0011 & 0111 0001	0001 0001
Logical OR	|	*Integer Bitwise*	`result = num1	num2;`	1001 0011 | 0111 0001	1111 0011
Exclusive OR	^	*Integer Bitwise*	`result = num1 ^ num2;`	1001 0011 ^ 0111 0001	1110 0010

Figure 2.6 Variable Operations

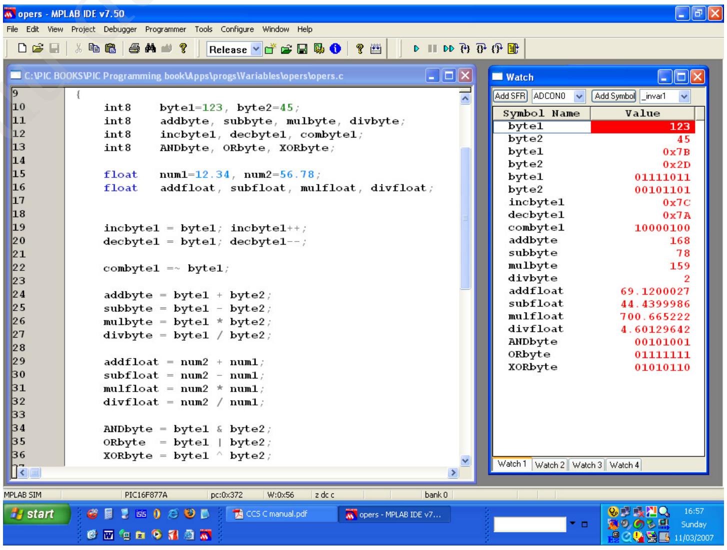


Table 2.7 Conditional Operators

Operation	Symbol	EXAMPLE
Equal to	==	`if(a == 0) b=b+5;`
Not equal to	!=	`if(a != 1) b=b+4;`
Greater than	>	`if(a > 2) b=b+3;`
Less than	<	`if(a < 3) b=b+2;`
Greater than or equal to	>=	`if(a >= 4) b=b+1;`
Less than or equal to	<=	`if(a <= 5) b=b+0;`

2.4 PIC16 C Sequence Control

- While loops
- Break, continue, goto
- If, else, switch

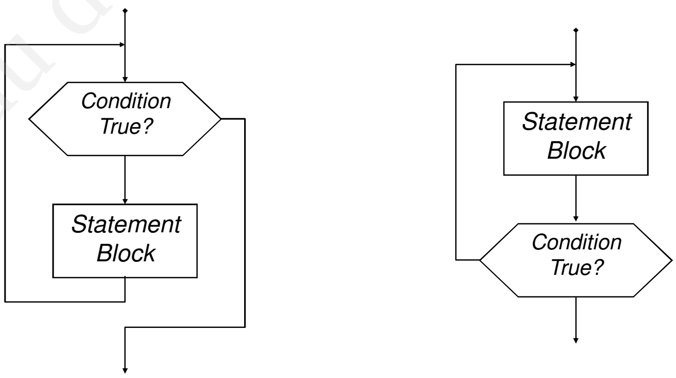
Conditional branching operations are a basic feature of any program. These must be properly organized so that the program structure is maintained and confusion avoided. The program then is easy to understand and more readily modified and upgraded.

While Loops

The basic ***while(condition)*** provides a logical test at the start of a loop, and the statement block is executed only if the condition is true. It may, however, be desirable that the loop block be executed at least once, particularly if the test condition is affected within the loop. This option is provided by the ***do..while(condition)*** syntax. The difference between these alternatives is illustrated in Figure 2.7 . The WHILE test occurs before the block and the DO WHILE after.

The program DOWHILE shown in Listing 2.9 includes the same block of statements contained within both types of loop. The WHILE block is not executed because the loop control variable has been set to 0 and is never modified. By contrast, ' count ' is incremented within the DO WHILE loop before being tested, and the loop therefore is executed.

Figure 2.3.1 Comparison of While and Do..While Loop



(a) While loop **(b) Do..While loop**

Listing 2.9 DOWHILE.C contains both types of 'while' loop

```
//   DOWHILE.C
//   Comparison of WHILE and DO WHILE loops

#include "16F877A.H"

main()
{
int outbyte1=0;
     int outbyte2=0;
     int count;

     count=0;                           // This loop is not
     while (count!=0)                   // executed
     {        output_C(outbyte1);
              outbyte1++;
              count--;
     }

     count=0;                           // This loop is
     do                                 // executed
     {        output_C(outbyte2);
              outbyte2++;
              count--;
     } while (count!=0);

     while(1){};
}
```

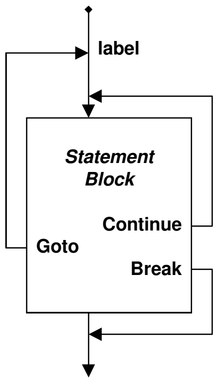
Break, Continue, and Goto

It may sometimes be necessary to ***break*** the execution of a loop or block in the middle of its sequence (Figure 2.8). The block must be exited in an orderly way, and it is useful to have the option of restarting the block (***continue***) or proceeding to the next one (***break***).

Occasionally, an unconditional jump may be needed, but this should be regarded as a last resort, as it tends to threaten the program stability. It is achieved by assigning a label to the jump destination and executing a ***goto***..label.

The use of these control statements is illustrated in Listing 2.10 . The events that trigger break and continue are asynchronous (independent of the program timing) inputs from external switches, which allows the counting loop to be quit or restarted at any time.

Figure 2.8 *Break, continue and goto*



Listing 2.10 *Continue, Break & Goto*

```
//      CONTINUE.C
//      Continue, break and goto jumps

#include "16F877A.H"
#use delay(clock=4000000)

main()
{
    int outbyte;

    again: outbyte=0;                           // Goto destination

    while(1)
    {
        output_C(outbyte);                      // Loop operation
        delay_ms(10);
        outbyte++;

        if (!input(PIN_D0)) continue;           // Restart loop
        if (!input(PIN_D1)) break;              // Terminate loop
        delay_ms(100);
        if (outbyte==100) goto again;           // Unconditional jump
    }
}
```

If..Else and Switch..Case

We have seen the basic if control option, which allows a block to be executed or skipped conditionally. The else option allows an alternate sequence to be executed, when the if block is skipped. We also need a multichoice selection, which is provided by the ***switch..case*** syntax. This tests a variable value and provides a set of alternative sequences, one of which is selected depending on the test result.
These options are illustrated in flowchart form in Figures 2.9 and 2.10 , and the ***if.. else*** and ***switch..case*** syntax is shown in Listing 2.11 . The control statement switch(variable) tests the value of the variable used to select the option block.
The keyword case n: is used to specify the value for each option. Note that each option block must be terminated with break, which causes the remaining blocks to be skipped.
A default block is executed if none of the options is taken. The same effect can be achieved using if..else, but switch..case provides a more elegant solution for implementing multichoice operations, such as menus.
If the case options comprise more than one statement, they are best implemented using a function block call, as explained in the next section.

Figure 2.9 Comparison of If and If..Else

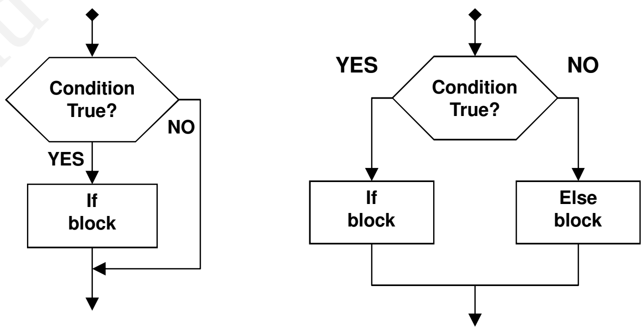
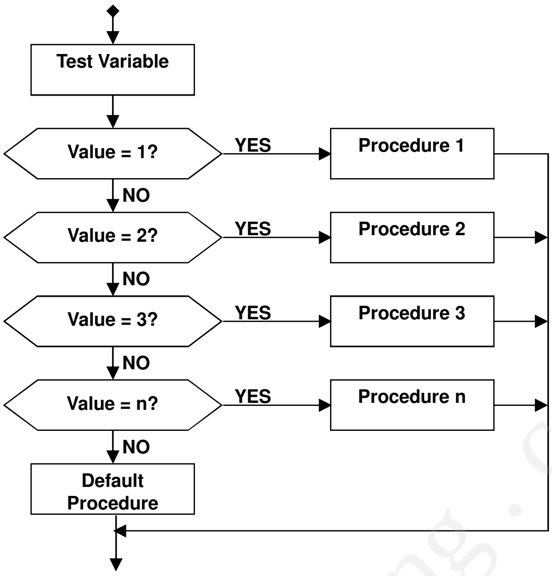


Figure 2.10 *Switch..case branching structure*



Listing 2.11 *Comparison of Switch and If..Else control*

```
//          SWITCH.C
//          Switch and if..else sequence control
//          Same result from both sequences

#include "16F877A.h"

void main()
{
    int8 inbits;

    while(1)
    {
        inbits = input_D();                     // Read input byte

// Switch..case option................................................

        switch(inbits)                          // Test input byte
        {
            case 1: output_C(1);                // Input = 0x01, output = 0x01
                    break;                      // Quit block
            case 2: output_C(3);                // Input = 0x02, output = 0x03
                    break;                      // Quit block
            case 3: output_C(7);                // Input = 0x03, output = 0x07
                    break;                      // Quit block
            default:output_C(0);                // If none of these, output = 0x00
        }

// If..else option...................................................

        if (input(PIN_D0))   output_C(1);                     // Input RD0 high
        if (input(PIN_D1))   output_C(2);                     // Input RD1 high
        if (input(PIN_D0) && input(PIN_D1)) output_C(7);      // Both high
        else output_C(0);                                     // If none of these, output = 0x00

    }
}
```

2.5 PIC16 C Functions and Structure

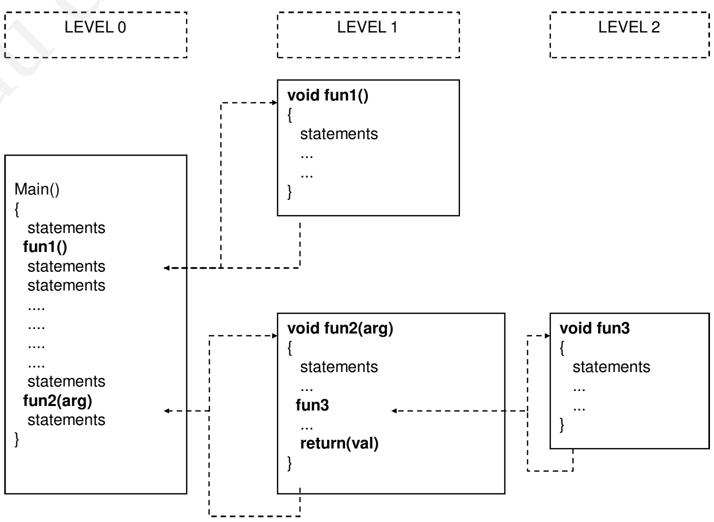
- Program structure
- Functions, arguments
- Global and local variables

The structure of a C program is created using functions (Figure 2.11). This is a block of code written and executed as a self-contained process, receiving the required parameters (data to be processed) from the calling function and returning results to it. Main() is the primary function in all C programs, within which the rest of the program is constructed.

When running on a PC, main() is called by the operating system, and control is returned to the OS when the C program is terminated. In the microcontroller, main() is simply used to indicate the start of the main control sequence, and more care needs to be taken in terminating the program.

Normally, the program runs in a continuous loop, but if not, the final statement should be while(1);, which causes the program to wait and prevents the program running into undefined locations following the application code.

Figure 2.11 Hierarchical C program structure



Basic Functions

A simple program using a function is shown in FUNC1.C, Listing 2.12 . The main block is very short, consisting of the function call out() and a while statement, which provides the wait state at the end of main(). In this case, the variables are declared *before the main block. This makes them global in scope; that is, they are recognized* throughout the whole program and within all function blocks. The function out() is also defined *before main() , so that, when it is called, the function name is recognized. The* function starts with the keyword void , which indicates that no value is returned by the function. The significance of this is explained shortly.
The function itself simply increments Port C from 0 to 255. It contains a for loop to provide a delay, so that the output count is visible. This is a simple alternative to the built-in delay functions seen in previous examples and is used here to avoid the inclusion of such functions while we study user-defined functions. It simply counts up to a preset value to waste time. The delay time is controlled by this set value.

Listing 2.12 Basic function call

```
//  FUNC1.C
//  Function call structure

#include "16F877A.H"

int8    outbyte=1;
int16   n;

void out()                  // Start of function block
{
    while (outbyte!=0)      // Start loop, quit when output =0
    {
            output_C(outbyte);     // Output code 1 - 0xFF
            outbyte++;             // Increment output
            for(n=1;n<500;n++);    // Delay so output is visible
    }
}

main()
{
    out();                  // Function call
    while(1);               // Wait until reset
}
```

Global and Local Variables

Now, assume that we wish to pass a value to the function for local use (that is, within the function). The simplest way is to define it as a global variable, which makes it available throughout the program. In program FUNC2.C, Listing 2.13 , the variable count, holding the delay count, hence the delay time, is global.

If there is no significant restriction on program memory, global variables may be used. However, microcontrollers, by definition, have limited memory, so it is desirable to use local variables whenever possible within the user functions. This is because local variables exist only during function execution, and the locations used for them are freed up on completion of function call. This can be confirmed by watching the values of C program variables when the program is executed in simulation mode — the local ones become undefined once the relevant function block is terminated.
If only global variables are used and the functions do not return results to the calling block, they become procedures. Program FUNC3.C, Listing 2.14 , shows how local variables are used.

Listing 2.13 Passing a parameter to the function

```
//  FUNC2.C

#include "16F877A.H"

int8    outbyte=1;              // Declare global variables
int16   n,count;

    void out()                  // Function block
    {
        while (outbyte!=0)
           {  output_C(outbyte);
              outbyte++;
              for(n=1;n<count;n++);
           }
    }

main()
{
    count=2000;
    out();                      // Call function
    while(1);
}
```

Listing 2.14 Local variables

```
//  FUNC3.C
//  Use of local variables

#include "16F877A.H"

int8    outbyte=1;              // Declare global variables
int16   count;

int out(int16 t)                // Declare argument types
{
    int16 n;                    // Declare local variable

    while (input(PIN_D0))       // Run output at speed t
    {     outbyte++;
          for(n=1;n<t;n++);
    }
    return outbyte;             // Return output when loop stops
}

main()
{
    count=50000;
    out(count);                 // Pass count value to function
    output_C(outbyte);          // Display returned value
    while(1);
}
```

2.6 PIC16 C Input and Output

- RS232 serial data
- Serial LCD
- Calculator and keypad

If an electronic gadget has a small alphanumeric LCD, the chances are that it is a microcontroller application. Smart card terminals, mobile phones, audio systems, coffee machines, and many other small systems use this display. The LCD we use here has a standard serial interface, and only one signal connection is needed. The signal format is RS232, a simple low-speed protocol that allows 1 byte or character code to be sent at a time. The data sequence also includes start and stop bits, and simple error checking can be applied if required. The PIC 16F877, in common with many microcontrollers, has a hardware RS232 port built in.

Serial LCD

CCS C provides an RS232 driver routine that works with any I/O pin (that is, the hardware port need not be used). This is possible because the process for generating the RS232 data frame is not too complex and can be completed fast enough to generate the signal in real time. At the standard rate of 9600 baud, each bit is about 100 μ s long, giving an overall frame time of about 1 ms. The data can be an 8-bit integer or, more often, a 7-bit ASCII character code. This method of transferring character codes via a serial line was originally used in mainframe computer terminals to send keystrokes to the computer and return the output — that is how long it's been around. In this example, the LCD receives character codes for a 2-row 16-character display.

The program uses library routines to generate the RS232 output, which are called up by the directive **# use RS232**. The baud rate must be specified and the send (TX) and receive (RX) pins specified as arguments of this directive. The directive must be preceded by a # use delay, which specifies the clock rate in the target system. The LCD has its own controller, which is compatible with the Hitachi 44780 MCU, the standard for this interface.
When the system is started, the LCD takes some time to initialize itself; its own MCU needs time to get ready to receive data. A delay of about 500 ms should be allowed in the main controller before attempting to access the LCD. A basic program for driving the LCD is shown in Listing 2.15 .

Listing 2.15 Serial LCD Operation

```
//  LCD.C
//  Serial LCD test-send character using putc() and printf()
///////////////////////////////////////////////////////////////

#include "16F877A.h"
#use delay(clock=4000000)
#use rs232(baud=9600, xmit=PIN_D0, rcv=PIN_D1)    // Define speed
                                                      and pins

void main()
{
  char acap='A';       // Test data

  delay_ms(1000);      // Wait for LCD to wake up
  putc(254); putc(1);  // Home cursor
  delay_ms(10);        // Wait for LCD to finish

  while(1)
  {
    putc(acap);                            // Send test character
    putc(254); putc(192); delay_ms(10);    // Move to second row
    printf("ASCII %c CHAR %d ",acap,acap); // Send test data again
    while(1);
  }
}
```

Table 2.8: Essential Control Codes for Serial 2x16 LCD

Code	Effect
254	Switch to control mode
followed by	
00	Home to start of row 1
01	Clear screen
192	Go to start of row 2

Table 2.9: Output Format Codes

Code	Displays
`%d`	Signed integer
`%u`	Unsigned integer
`%Lu`	Long unsigned integer (16 or 32 bits)
`%Ls`	Long signed integer (16 or 32 bits)
`%g`	Rounded decimal float (use decimal formatting)
`%f`	Truncated decimal float (use decimal formatting)
`%e`	Exponential form of float
`%w`	Unsigned integer with decimal point inserted (use decimal formatting)
`%X`	Hexadecimal
`%LX`	Long hex
`%c`	ASCII character corresponding to numerical value
`%s`	Character or string

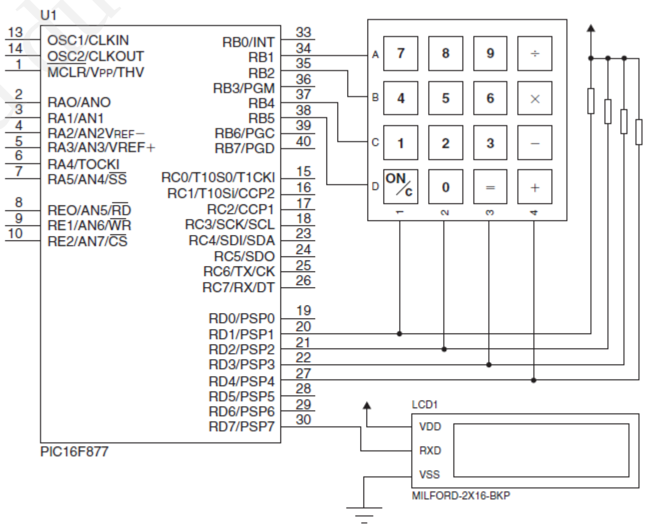


Figure 2.12: Calculator Schematic

2.7 PIC16 C More Data Types

- Arrays and strings
- Pointers and indirect addressing
- Enumeration

The data in a C program may be most conveniently handled as sets of associated variables. These occur more frequently as the program data becomes more complex, but only the basics are mentioned here.

Listing 2.18 Numerical and Character Arrays

```
// ARRAYS.C
// Demo of numerical and string arrays
// Attach ARRAYS.COF to LCD.DSN to display
//////////////////////////////////////////////////////////////
#include "16F877A.h"
#use delay(clock=4000000)
#use rs232(baud=9600, xmit=PIN_D0, rcv=PIN_D1)

main()
{
  int8 aval=0, n;    // Declare single variables
  int8 anum[10];     // Declare integer array
  char astring[16];  // Declare character array

// Start LCD............................................
  delay_ms(1000);
  putc(254); putc(1); delay_ms(10);

// Assign data to arrays..................................
  for ( n=0; n<10; n++ ) { anum[n]=aval; aval++; }
  strcpy(astring,"Hello!");

// Display data..........................................
  for ( n=0; n<10; n++ ) printf("%d",anum[n]);
  putc(254); putc(192); delay_ms(10);
  puts(astring);

  while(1);    // Wait
}
```

2.8 PIC16 C Compiler Directives

- Include and use directives
- Header file listing and directives

Compiler directives are typically used at the top of the program to set up compiler options, control project components, define constant labels, and so on before the main program is created. They are preceded by the hash symbol to distinguish them from other types of statements and do not have a semicolon to end the line.

Program Directives

Examples using the directives encountered thus far follow—refer to the compiler reference manual for the full range of options.

```
#include "16F877A.h"
```

The include directive allows source code files to be included as though they had been typed in by the user. In fact, any block of source code can be included in this way, and the directive can thus be used to incorporate previously written reusable functions. The header file referred to in this case provides the information needed by the complier to create a program for a specific PIC chip.

```
#use delay(clock=4000000)
```

The 'use' directive allows library files to be included. As can be seen, additional operating parameters may be needed so that the library function works correctly. The clock frequency given here needs to be specified so that both software and hardware timing loops can be correctly calculated.

```
#use rs232(baud=9600, xmit=PIN_D0, rcv=PIN_D1)
```

In this directive, the parameters set the RS232 data (baud) rate and the MCU pins to be used to transmit and receive the signal. This software serial driver allows any available pin to be used.

Listing 2.19 Header File 16F877A.H

```
//////// Standard Header file for the PIC16F877A device /////////
#device PIC16F877A
#nolist
//////// Program memory: 8192x14 Data RAM: 367 Stack: 8
//////// I/O: 33 Analog Pins: 8
//////// Data EEPROM: 256
//////// C Scratch area: 77 ID Location: 2000
//////// Fuses: LP,XT,HS,RC,NOWDT,WDT,NOPUT,PUT,PROTECT,DEBUG,NODEBUG
//////// Fuses: NOPROTECT,NOBROWNOUT,BROWNOUT,LVP,NOLVP,CPD,NOCPD,WRT_50%
//////// Fuses: NOWRT,WRT_25%,WRT_5%
////////
////////////////////////////////////////////////////////////////////////
//
// Discrete I/O Functions: SET_TRIS_x(), OUTPUT_x(), INPUT_x(),
//                         PORT_B_PULLUPS(), INPUT(),
//                         OUTPUT_LOW(), OUTPUT_HIGH(),
//                         OUTPUT_FLOAT(), OUTPUT_BIT()
//
// Constants used to identify pins in the above are:
#define PIN_A0 40   // Register 05, pin 0 (5x8)+0=40
#define PIN_A1 41   // Register 05, pin 1 (5x8)+1=41
#define PIN_A2 42   // Register 05, pin 2 (5x8)+2=42
#define PIN_A3 43   // Register 05, pin 3 etc
#define PIN_A4 44   // Register 05, pin 4
#define PIN_A5 45   // Register 05, pin 5
```

```
#define PIN_B0 48   // Register 06, pin 0 (6*8)+0=48
#define PIN_B1 49   // Register 06, pin 1 etc
#define PIN_B2 50   // Register 06, pin 2
#define PIN_B3 51   // Register 06, pin 3
#define PIN_B4 52   // Register 06, pin 4
#define PIN_B5 53   // Register 06, pin 5
#define PIN_B6 54   // Register 06, pin 6
#define PIN_B7 55   // Register 06, pin 7

#define PIN_C0 56   // Register 07, pin 0 (7*8)+0=56
#define PIN_C1 57   // Register 07, pin 1 etc
#define PIN_C2 58   // Register 07, pin 2
#define PIN_C3 59   // Register 07, pin 3
#define PIN_C4 60   // Register 07, pin 4
#define PIN_C5 61   // Register 07, pin 5
#define PIN_C6 62   // Register 07, pin 6
#define PIN_C7 63   // Register 07, pin 7
```

```
#define PIN_D0 64           // Register 08, pin 0 (8*8)+0=64
#define PIN_D1 65           // Register 08, pin 1 etc
#define PIN_D2 66           // Register 08, pin 2
#define PIN_D3 67           // Register 08, pin 3
#define PIN_D4 68           // Register 08, pin 4
#define PIN_D5 69           // Register 08, pin 5
#define PIN_D6 70           // Register 08, pin 6
#define PIN_D7 71           // Register 08, pin 7

#define PIN_E0 72           // Register 09, pin 0 (9*8)+0=72
#define PIN_E1 73           // Register 09, pin 1 etc
#define PIN_E2 74           // Register 09, pin 2

///////////////////////////////////////////////////////// Useful defines

#define FALSE 0             // Logical state 0
#define TRUE 1              // Logical state 1

#define BYTE int            // 8-bit value
#define BOOLEAN short int   // 1-bit value

#define getc getch          // Alternate names..
#define fgetc getch         // ..for identical functions
#define getchar getch
#define putc putchar
#define fputc putchar
#define fgets gets
#define fputs puts
```

```
/////////////////////////////////////////////////////////////// Control
// Control Functions: RESET_CPU(), SLEEP(), RESTART_CAUSE()
// Constants returned from RESTART_CAUSE() are:
#define WDT_FROM_SLEEP  0    // Watchdog timer has woken MCU from sleep
#define WDT_TIMEOUT     8    // Watchdog timer has caused reset
#define MCLR_FROM_SLEEP 16   // MCU has been woken by reset input
#define NORMAL_POWER_UP 24   // Normal power on reset has occurred

/////////////////////////////////////////////////////////////// Timer 0
// Timer 0 (AKA RTCC)Functions: SETUP_COUNTERS() or SETUP_TIMER0(),
//                              SET_TIMER0() or SET_RTCC(),
//                              GET_TIMER0() or GET_RTCC()

// Constants used for SETUP_TIMER0() are:
#define RTCC_INTERNAL   0    // Use instruction clock
#define RTCC_EXT_L_TO_H 32   // Use T0CKI rising edge
#define RTCC_EXT_H_TO_L 48   // Use T0CKI falling edge

#define RTCC_DIV_1      8    // No prescale
#define RTCC_DIV_2      0    // Prescale divide by 2
#define RTCC_DIV_4      1    // Prescale divide by 4
```

```
#define RTCC_DIV_8    2      // Prescale divide by 8
#define RTCC_DIV_16   3      // Prescale divide by 16
#define RTCC_DIV_32   4      // Prescale divide by 32
#define RTCC_DIV_64   5      // Prescale divide by 64
#define RTCC_DIV_128  6      // Prescale divide by 128
#define RTCC_DIV_256  7      // Prescale divide by 256

#define RTCC_8_BIT 0

// Constants used for SETUP_COUNTERS() are the above
// constants for the 1st param and the following for
// the 2nd param:

////////////////////////////////////////////////////////////////// WDT
// Watch Dog Timer Functions: SETUP_WDT() or SETUP_COUNTERS() (see above)
//                             RESTART_WDT()

// Constants used for SETUP_WDT() are:
#define WDT_18MS      8      // Watchdog timer interval=18ms
#define WDT_36MS      9      // Watchdog timer interval=36ms
#define WDT_72MS      10     // Watchdog timer interval=72ms
#define WDT_144MS     11     // Watchdog timer interval=144ms
#define WDT_288MS     12     // Watchdog timer interval=288s
#define WDT_576MS     13     // Watchdog timer interval=576ms
#define WDT_1152MS    14     // Watchdog timer interval=1.15ms
#define WDT_2304MS    15     // Watchdog timer interval=2.30s
```

```
///////////////////////////////////////////////////////////////// Timer1
// Timer 1 Functions: SETUP_TIMER_1, GET_TIMER1, SET_TIMER1

// Constants used for SETUP_TIMER_1() are:
//     (or (via |) together constants from each group)
#define T1_DISABLED        0       // Switch off Timer 1
#define T1_INTERNAL        0x85    // Use instruction clock
#define T1_EXTERNAL        0x87    // Use T1CKI as clock input
#define T1_EXTERNAL_SYNC   0x83    // Synchronise T1CKI input
#define T1_CLK_OUT         8
#define T1_DIV_BY_1        0       // No prescale
#define T1_DIV_BY_2        0x10    // Prescale divide by 2
#define T1_DIV_BY_4        0x20    // Prescale divide by 4
#define T1_DIV_BY_8        0x30    // Prescale divide by 8

///////////////////////////////////////////////////////////////// Timer 2
// Timer 2 Functions: SETUP_TIMER_2, GET_TIMER2, SET_TIMER2
// Constants used for SETUP_TIMER_2() are:
#define T2_DISABLED        0       // No prescale
#define T2_DIV_BY_1        4       // Prescale divide by 2
#define T2_DIV_BY_4        5       // Prescale divide by 4
#define T2_DIV_BY_16       6       // Prescale divide by 16
```